In general, convex sets may have either straight or curved boundaries and may be bounded or unbounded. Convex sets may be topologically open or closed. Some examples are shown in the figure below. The convex hull of a finite set of points in the plane is a bounded, closed, convex polygon.

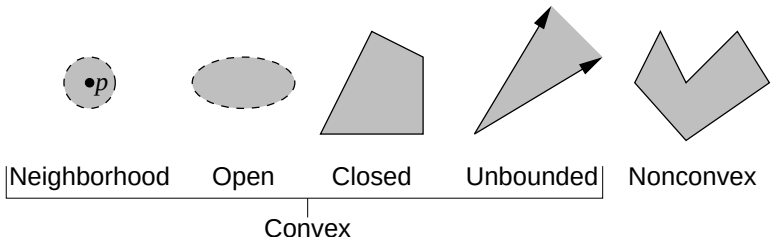

Figure 10: Terminology.

Convex hull problem: The (planar) *convex hull problem* is, given a set of n points P in the plane, output a representation of P's convex hull. The convex hull is a closed convex polygon, the simplest representation is a counterclockwise enumeration of the vertices of the convex hull. (A clockwise is also possible. We usually prefer counterclockwise enumerations, since they correspond to positive orientations, but obviously one representation is easily converted into the other.) Ideally, the hull should consist only of *extreme points*, in the sense that if three points lie on an edge of the boundary of the convex hull, then the middle point should not be output as part of the hull.

There is a simple $O(n^3)$ convex hull algorithm, which operates by considering each ordered pair of points (p, q), and the determining whether all the remaining points of the set lie within the half-plane lying to the right of the directed line from p to q. (Observe that this can be tested using the orientation test.) The question is, can we do better?

Graham's scan: We will present an $O(n \log n)$ algorithm for convex hulls. It is a simple variation of a famous algorithm for convex hulls, called *Graham's scan*. This algorithm dates back to the early 70's. The algorithm is based on an approach called *incremental construction*, in which points are added one at a time, and the hull is updated with each new insertion. If we were to add points in some arbitrary order, we would need some method of testing whether points are inside the existing hull or not. To avoid the need for this test, we will add points in increasing order of x-coordinate, thus guaranteeing that each newly added point is outside the current hull. (Note that Graham's original algorithm sorted points in a different way. It found the lowest point in the data set and then sorted points cyclically around this point.)

Since we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right. The convex hull is a cyclically ordered sets. Cyclically ordered sets are somewhat messier to work with than simple linearly ordered sets, so we will break the hull into two hulls, an *upper hull* and *lower hull*. The break points common to both hulls will be the leftmost and rightmost vertices of the convex hull. After building both, the two hulls can be concatenated into a single cyclic counterclockwise list.

Here is a brief presentation of the algorithm for computing the upper hull. We will store the hull vertices in a stack U, where the top of the stack corresponds to the most recently added point. Let first(U) and second(U) denote the top and second element from the top of U, respectively. Observe that as we read the stack from top to bottom, the points should make a (strict) left-hand turn, that is, they should have a positive orientation. Thus, after adding the last point, if the previous two points fail to have a positive orientation, we pop them off the stack. Since the orientations of remaining points on the stack are unaffected, there is no need to check any points other than the most recent point and its top two neighbors on the stack.

Let us consider the upper hull, since the lower hull is symmetric. Let $\langle p_1, p_2, \ldots, p_n \rangle$ denote the set of points, sorted by increase x-coordinates. As we walk around the upper hull from left to right, observe that each consecutive triple along the hull makes a right-hand turn. That is, if p, q, r are consecutive points along the upper hull, then Orient$(p, q, r) < 0$. When a new point p_i is added to the current hull, this may violate the right-hand turn

(1) Sort the points according to increasing order of their x-coordinates, denoted $\langle p_1, p_2, \ldots, p_n \rangle$.

(2) Push p_1 and then p_2 onto U.

(3) for $i = 3$ to n do:

  (a) while size$(U) \geq 2$ and Orient$(p_i, \text{first}(U), \text{second}(U)) \leq 0$, pop U.

  (b) Push p_i onto U.

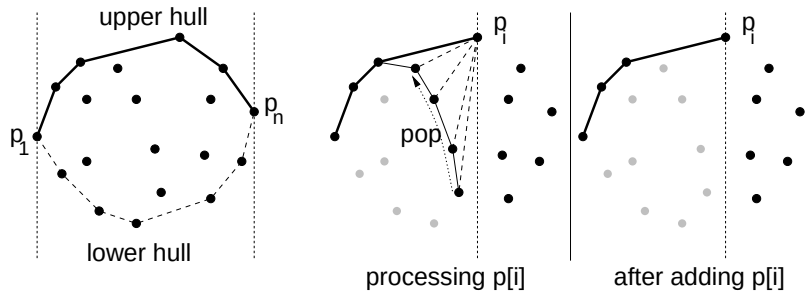

Figure 11: Convex hulls and Graham's scan.

invariant. So we check the last three points on the upper hull, including p_i. They fail to form a right-hand turn, then we delete the point prior to p_i. This is repeated until the number of points on the upper hull (including p_i) is less than three, or the right-hand turn condition is reestablished. See the text for a complete description of the code. We have ignored a number of special cases. We will consider these next time.

Analysis: Let us prove the main result about the running time of Graham's scan.

Theorem: Graham's scan runs in $O(n \log n)$ time.

Proof: Sorting the points according to x-coordinates can be done by any efficient sorting algorithm in $O(n \log n)$ time. Let D_i denote the number of points that are popped (deleted) on processing p_i. Because each orientation test takes $O(1)$ time, the amount of time spent processing p_i is $O(D_i + 1)$. (The extra $+1$ is for the last point tested, which is not deleted.) Thus, the total running time is proportional to

$$\sum_{i=1}^{n} (D_i + 1) \;=\; n + \sum_{i=1}^{n} D_i.$$

To bound $\sum_i D_i$, observe that each of the n points is pushed onto the stack once. Once a point is deleted it can never be deleted again. Since each of n points can be deleted at most once, $\sum_i D_i \leq n$. Thus after sorting, the total running time is $O(n)$. Since this is true for the lower hull as well, the total time is $O(2n) = O(n)$.

Convex Hull by Divide-and-Conquer: As with sorting, there are many different approaches to solving the convex hull problem for a planar point set P. Next we will consider another $O(n \log n)$ algorithm, which is based on the divide-and-conquer design technique. It can be viewed as a generalization of the famous MergeSort sorting algorithm (see Cormen, Leiserson, and Rivest). Here is an outline of the algorithm. It begins by sorting the points by their x-coordinate, in $O(n \log n)$ time.

The asymptotic running time of the algorithm can be expressed by a recurrence. Given an input of size n, consider the time needed to perform all the parts of the procedure, ignoring the recursive calls. This includes the time to partition the point set, compute the two tangents, and return the final result. Clearly the first and third of these steps can be performed in $O(n)$ time, assuming a linked list representation of the hull vertices. Below we

(1) If $|P| \leq 3$, then compute the convex hull by brute force in $O(1)$ time and return.

(2) Otherwise, partition the point set P into two sets A and B, where A consists of half the points with the lowest x-coordinates and B consists of half of the points with the highest x-coordinates.

(3) Recursively compute $H_A = \mathcal{CH}(A)$ and $H_B = \mathcal{CH}(B)$.

(4) Merge the two hulls into a common convex hull, H, by computing the upper and lower tangents for H_A and H_B and discarding all the points lying between these two tangents.

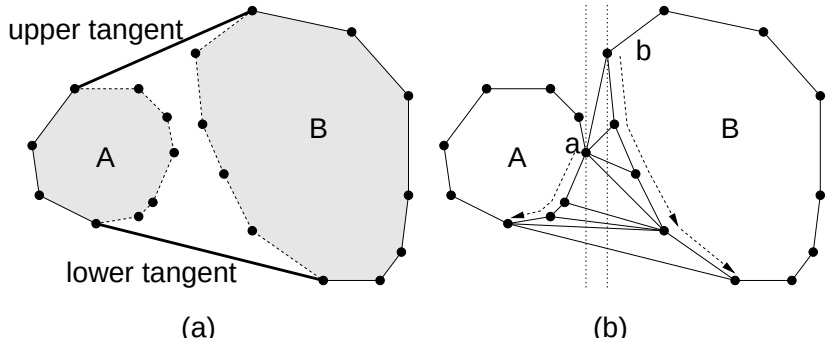

Figure 12: Computing the lower tangent.

will show that the tangents can be computed in $O(n)$ time. Thus, ignoring constant factors, we can describe the running time by the following recurrence.

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ n + 2T(n/2) & \text{otherwise.} \end{cases}$$

This is the same recurrence that arises in Mergesort. It is easy to show that it solves to $T(n) \in O(n \log n)$ (see CLR). All that remains is showing how to compute the two tangents.

One thing that simplifies the process of computing the tangents is that the two point sets A and B are separated from each other by a vertical line (assuming no duplicate x-coordinates). Let's concentrate on the lower tangent, since the upper tangent is symmetric. The algorithm operates by a simple "walking" procedure. We initialize a to be the rightmost point of H_A and b is the leftmost point of H_B. (These can be found in linear time.) Lower tangency is a condition that can be tested locally by an orientation test of the two vertices and neighboring vertices on the hull. (This is a simple exercise.) We iterate the following two loops, which march a and b down, until they reach the points lower tangency.

Finding the Lower Tangent

LowerTangent(H_A, H_B) :

(1) Let a be the rightmost point of H_A.

(2) Let b be the leftmost point of H_B.

(3) While ab is not a lower tangent for H_A and H_B do

  (a) While ab is not a lower tangent to H_A do $a = a - 1$ (move a clockwise).

  (b) While ab is not a lower tangent to H_B do $b = b + 1$ (move b counterclockwise).

(4) Return ab.

Proving the correctness of this procedure is a little tricky, but not too bad. Check O'Rourke's book out for a careful proof. The important thing is that each vertex on each hull can be visited at most once by the search, and

hence its running time is $O(m)$, where $m = |H_A| + |H_B| \leq |A| + |B|$. This is exactly what we needed to get the overall $O(n \log n)$ running time.

Lecture 4: More Convex Hull Algorithms

Reading: Today's material is not covered in the 4M's book. It is covered in O'Rourke's book on Computational Geometry. Chan's algorithm can be found in T. Chan, "Optimal output-sensitive convex hull algorithms in two and three dimensions", *Discrete and Computational Geometry*, 16, 1996, 361–368.

QuickHull: If the divide-and-conquer algorithm can be viewed as a sort of generalization of MergeSort, one might ask whether there is corresponding generalization of other sorting algorithm for computing convex hulls. In particular, the next algorithm that we will consider can be thought of as a generalization of the QuickSort sorting procedure. The resulting algorithm is called QuickHull.

Like QuickSort, this algorithm runs in $O(n \log n)$ time for favorable inputs but can take as long as $O(n^2)$ time for unfavorable inputs. However, unlike QuickSort, there is no obvious way to convert it into a randomized algorithm with $O(n \log n)$ expected running time. Nonetheless, QuickHull tends to perform very well in practice.

The intuition is that in many applications most of the points lie in the interior of the hull. For example, if the points are uniformly distributed in a unit square, then it can be shown that the expected number of points on the convex hull is $O(\log n)$.

The idea behind QuickHull is to discard points that are not on the hull as quickly as possible. QuickHull begins by computing the points with the maximum and minimum, x- and y-coordinates. Clearly these points must be on the hull. Horizontal and vertical lines passing through these points are support lines for the hull, and so define a bounding rectangle, within which the hull is contained. Furthermore, the convex quadrilateral defined by these four points lies within the convex hull, so the points lying within this quadrilateral can be eliminated from further consideration. All of this can be done in $O(n)$ time.

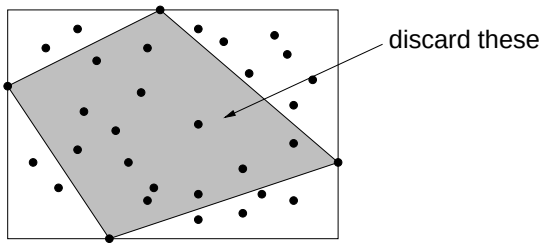

Figure 13: QuickHull's initial quadrilateral.

To continue the algorithm, we classify the remaining points into the four corner triangles that remain. In general, as this algorithm executes, we will have an inner convex polygon, and associated with each edge we have a set of points that lie "outside" of that edge. (More formally, these points are witnesses to the fact that this edge is not on the convex hull, because they lie outside the half-plane defined by this edge.) When this set of points is empty, the edge is a final edge of the hull. Consider some edge ab. Assume that the points that lie "outside" of this hull edge have been placed in a bucket that is associated with ab. Our job is to find a point c among these points that lies on the hull, discard the points in the triangle abc, and split the remaining points into two subsets, those that lie outside ac and those than lie outside of cb. We can classify each point by making two orientation tests.

How should c be selected? There are a number of possible selection criteria that one might think of. The method that is most often proposed is to let c be the point that maximizes the perpendicular distance from the line ab. (For example, another possible choice might be the point that maximizes the angle cba or cab. It turns out that

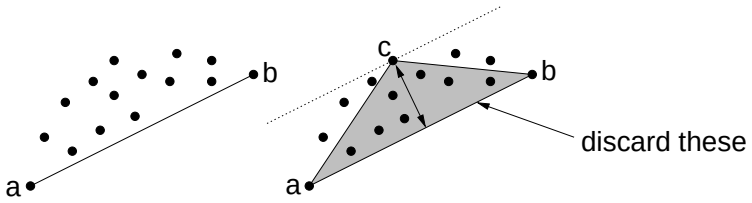

Figure 14: QuickHull elimination procedure.

these can be are very poor choices because they tend to produce imbalanced partitions of the remaining points.) We replace the edge ab with the two edges ac and cb, and classify the points as lying in one of three groups: those that lie in the triangle abc, which are discarded, those that lie outside of ac, and those that lie outside of cb. We put these points in buckets for these edges, and recurse. (We claim that it is not hard to classify each point p, by computing the orientations of the triples acp and cbp.)

The running time of Quickhull, as with QuickSort, depends on how evenly the points are split at each stage. Let $T(n)$ denote the running time on the algorithm assuming that n points remain outside of some edge. In $O(n)$ time we can select a candidate splitting point c and classify the points in the bucket in $O(n)$ time. Let n_1 and n_2 denote the number of remaining points, where $n_1 + n_2 \leq n$. Then the running time is given by the recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n_1) + T(n_2) & \text{where } n_1 + n_2 \leq n. \end{cases}$$

In order to solve this recurrence, it would be necessary to determine the "reasonable" values for n_1 and n_2. If we assume that the points are "evenly" distributed, in the sense that $\max(n_1, n_2) \leq \alpha n$ for some constant $\alpha < 1$, then by applying the same analysis as that used in QuickSort (see Cormen, Leiserson, Rivest) the running time will solve to $O(n \log n)$, where the constant factor depends on α. On the other hand, if the splits are not balanced, then the running time can easily increase to $O(n^2)$.

Does QuickHull outperform Graham's scan? This depends to a great extent on the distribution of the point set. There are variations of QuickHull that are designed for specific point distributions (e.g. points uniformly distributed in a square) and their authors claim that they manage to eliminate almost all of the points in a matter of only a few iterations.

Gift-Wrapping and Jarvis's March: The next algorithm that we will consider is a variant on an $O(n^2)$ sorting algorithm called SelectionSort. For sorting, this algorithm repeatedly finds the next element to add to the sorted order from the remaining items. The corresponding convex hull algorithm is called *Jarvis's march*. which builds the hull in $O(nh)$ time by a process called "gift-wrapping". The algorithm operates by considering any one point that is on the hull, say, the lowest point. We then find the "next" edge on the hull in counterclockwise order. Assuming that p_k and p_{k-1} were the last two points added to the hull, compute the point q that maximizes the angle $\angle p_{k-1} p_k q$. Thus, we can find the point q in $O(n)$ time. After repeating this h times, we will return back to the starting point and we are done. Thus, the overall running time is $O(nh)$. Note that if h is $o(\log n)$ (asymptotically smaller than $\log n$) then this is a better method than Graham's algorithm.

One technical detail is that when we to find an edge from which to start. One easy way to do this is to let p_1 be the point with the lowest y-coordinate, and let p_0 be the point $(-\infty, 0)$, which is infinitely far to the right. The point p_0 is only used for computing the initial angles, after which it is discarded.

Output Sensitive Convex Hull Algorithms: It turns out that in the worst-case, convex hulls cannot be computed faster than in $\Omega(n \log n)$ time. One way to see this intuitively is to observe that the convex hull itself is sorted along its boundary, and hence if every point lies on the hull, then computing the hull requires sorting of some form. Yao proved the much harder result that determining which points are on the hull (without sorting them along the boundary) still requires $\Omega(n \log n)$ time. However both of these results rely on the fact that all (or at least a constant fraction) of the points lie on the convex hull. This is often not true in practice.

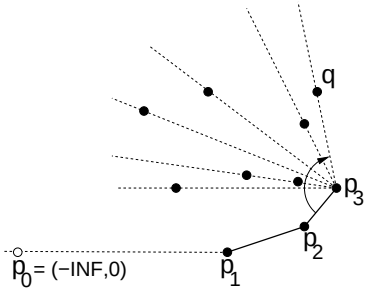

Figure 15: Jarvis's march.

The QuickHull and Jarvis's March algorithms that we saw last time suggest the question of how fast can convex hulls be computed if we allow the running time to be described in terms of both the input size n and the output size h. Many geometric algorithms have the property that the output size can be a widely varying function of the input size, and worst-case output size may not be a good indicator of what happens typically. An algorithm which attempts to be more efficient for small output sizes is called an *output sensitive algorithm*, and running time is described as a asymptotic function of both input size and output size.

Chan's Algorithm: Given than any convex hull algorithm must take at least $O(n)$ time, and given that "$\log n$" factor arises from the fact that you need to sort the at most n points on the hull, if you were told that there are only h points on the hull, then a reasonable target running time is $O(n \log h)$. (Below we will see that this is optimal.) Kirkpatrick and Seidel discovered a relatively complicated $O(n \log h)$ time algorithm, based on a clever pruning method in 1986. The problem was considered closed until around 10 years later when Timothy Chan came up with a much simpler algorithm with the same running time. One of the interesting aspects of Chan's algorithm is that it involves combining two slower algorithms (Graham's scan and Jarvis's March) together to form an algorithm that is faster than either one.

The problem with Graham's scan is that it sorts all the points, and hence is doomed to having an $\Omega(n \log n)$ running time, irrespective of the size of the hull. On the other hand, Jarvis's march can perform better if you have few vertices on the hull, but it takes $\Omega(n)$ time for each hull vertex.

Chan's idea was to partition the points into groups of equal size. There are m points in each group, and so the number of groups is $r = \lceil n/m \rceil$. For each group we compute its hull using Graham's scan, which takes $O(m \log m)$ time per group, for a total time of $O(rm \log m) = O(n \log m)$. Next, we run Jarvis's march on the groups. Here we take advantage of the fact that you can compute the tangent between a point and a convex m-gon in $O(\log m)$ time. (We will leave this as an exercise.) So, as before there are h steps of Jarvis's march, but because we are applying it to r convex hulls, each step takes only $O(r \log m)$ time, for a total of $O(hr \log m) = ((hn/m) \log m)$ time. Combining these two parts, we get a total of

$$O\left(\left(n + \frac{hn}{m}\right) \log m\right)$$

time. Observe that if we set $m = h$ then the total running time will be $O(n \log h)$, as desired.

There is only one small problem here. We do not know what h is in advance, and therefore we do not know what m should be when running the algorithm. We will see how to remedy this later. For now, let's imagine that someone tells us the value of m. The following algorithm works correctly as long as $m \geq h$. If we are wrong, it returns a special error status.

We assume that we store the convex hulls from step (2a) in an ordered array so that the step inside the for-loop of step (4a) can be solved in $O(\log m)$ time using binary search. Otherwise, the analysis follows directly from the comments made earlier.

Chan's Partial Convex Hull Algorithm (Given m)

PartialHull(P, m) :

(1) Let $r = \lceil n/m \rceil$. Partition P into disjoint subsets $P_1, P_2, \ldots, P_r$, each of size at most m.

(2) For $i = 1$ to r do:

  (a) Compute $\text{Hull}(P_i)$ using Graham's scan and store the vertices in an ordered array.

(3) Let $p_0 = (-\infty, 0)$ and let p_1 be the bottommost point of P.

(4) For $k = 1$ to m do:

  (a) For $i = 1$ to r do:

    - Compute point $q_i \in P_i$ that maximizes the angle $\angle p_{k-1} p_k q_i$.

  (b) Let p_{k+1} be the point $q \in \{q_1, \ldots, q_r\}$ that maximizes the angle $\angle p_{k-1} p_k q$.

  (c) If $p_{k+1} = p_1$ then return $\langle p_1, \ldots, p_k \rangle$.

(5) Return "m was too small, try again."

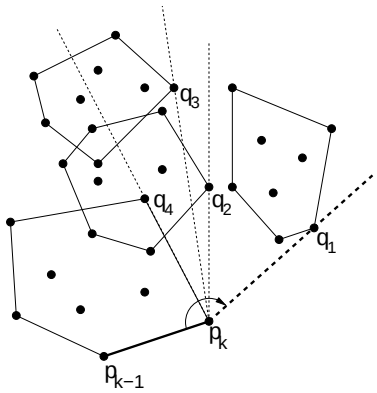

Figure 16: Chan's Convex Hull Algorithm.

The only question remaining is how do we know what value to give to m? We could try $m = 1, 2, 3, \ldots$, until we luck out and have $m \geq h$, but this would take too long. Binary search would be a more efficient option, but if we guess to large a value for m (e.g. $m = n/2$) then we are immediately stuck with $O(n \log n)$ time, and this is too slow.

Instead, the trick is to start with a small value of m and increase it rapidly. Since the dependence on m is only in the log term, as long as our value of m is within a polynomial of h, that is, $m = h^c$ for some constant c, then the running time will still be $O(n \log h)$. So, our approach will be to guess successively larger values of m, each time squaring the previous value, until the algorithm returns a successful result. This technique is often called *doubling search* (because the unknown parameter is successively doubled), but in this case we will be squaring rather than doubling.

Chan's Complete Convex Hull Algorithm

Hull(P) :

(1) For $t = 1, 2, \ldots$ do:
  (a) Let $m = \min(2^{2^t}, n)$.
  (b) Invoke PartialHull(P, m), returning the result in L.
  (c) If $L \neq$ "try again" then return L.

Note that 2^{2^t} has the effect of squaring the previous value of m. How long does this take? The t-th iteration takes $O(n \log 2^{2^t}) = O(n2^t)$ time. We know that it will stop as soon as $2^{2^t} \geq h$, that is if $t = \lceil \lg \lg n \rceil$. (We will use lg to denote logarithm base 2.) So the total running time (ignoring the constant factors) is

$$\sum_{t=1}^{\lg \lg h} n2^t \;=\; n \sum_{t=1}^{\lg \lg h} 2^t \;\leq\; n2^{1+\lg \lg h} \;=\; 2n \lg h \;=\; O(n \log h),$$

which is just what we want.

Lecture 5: Line Segment Intersection

Reading: Chapter 2 in the 4M's.

Geometric intersections: One of the most basic problems in computational geometry is that of computing intersections. Intersection computation in 2- and 3-space is basic to many different application areas.

- In solid modeling people often build up complex shapes by applying various boolean operations (intersection, union, and difference) to simple primitive shapes. The process in called *constructive solid geometry* (CSG). In order to perform these operations, the most basic step is determining the points where the boundaries of the two objects intersect.
- In robotics and motion planning it is important to know when two objects intersect for *collision detection* and *collision avoidance*.
- In geographic information systems it is often useful to *overlay* two subdivisions (e.g. a road network and county boundaries to determine where road maintenance responsibilities lie). Since these networks are formed from collections of line segments, this generates a problem of determining intersections of line segments.
- In computer graphics, *ray shooting* is an important method for rendering scenes. The computationally most intensive part of ray shooting is determining the intersection of the ray with other objects.

Most complex intersection problems are broken down to successively simpler and simpler intersection problems. Today, we will discuss the most basic algorithm, upon which most complex algorithms are based.